John von Neumann Institute for Computing

A New Model for NoC-based Distributed Heterogeneous System Design

F. Rincón, F. Moya, J. Barba, D. Villa, F.J. Villanueva, J.C. López

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 777-784, 2006.

http://www.fz-juelich.de/nic-series/volume33

A New Model for NoC-based Distributed Heterogeneous System Design*

F. Rincón[a], F. Moya[a], J. Barba[a], D. Villa[a], F. J. Villanueva[a] and J. C. López[a]

[a] School of Computer Science, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13071 Ciudad Real, Spain

This work explores the capabilities of a new design methodology aimed at offering an integrated and homogeneous way of thinking on the whole system. This methodology allows extending a widespread communication model, the remote method invocation model, to the design of a NoC system as a part of a more complex distributed system.

Some standardization efforts, such as OCP, have tried to define a common syntax for communications among NoC components but there is not a provision for a common semantics yet. On the other hand, there is a dramatic variation of the communication capabilities between two cores depending on the relative location of the two peers (on- and off- chip). Since a common communications infrastructure is missing, on-chip functionality may only be accessed from off-chip components using ad-hoc interfaces that exists only if it has been foreseen by the designer. In this work we will discuss the way our proposed methodology is able to tackle these and many other issues without major overhead.

1. Introduction

The Network on Chip (NoC) design paradigm, as an enabling technology for the integration of a high number of computational blocks interacting with each other, provides the adequate way to face an important part of the design of these applications. Taking the network as a facilitator to overcome complexity and scalability, NoCs face the main problems that arise when designing complex SoCs (System on Chip) composed of dozens, maybe hundreds, of IPs communicating with each other.

But these NoCs must interact with many other systems implemented in a vast range of technologies and using again the network as the basis to reach the goals of the new age applications: ubiquitous computing, cooperation between applications, knowledge management, multimedia communication, intelligent and sustainable growth… Besides the heterogeneity of the different system components, the resulting systems use also different and heterogeneous means of communication. A new concept appears, the distributed heterogeneous system, where the components are defined by the service they offer to the rest of the system, independently of their implementation and their location. This way of thinking on the functionality of a system poses new design challenges that claim for new design methodologies able to face them, keeping always in mind the reduction of the design time.

This work explores the capabilities of a new design methodology based on these concepts and able to offer an integrated and homogeneous way of thinking on the whole system. This methodology allows extending the remote method invocation model, to the design of a NoC system as a part of a more complex distributed system. Some standardization efforts, such as OCP[1], have tried to overcome the interface compatibility problem defining a common syntax for communications. But there is not a provision for a common semantics yet. On the other hand, there is a dramatic variation of the communication capabilities between two cores depending on the relative location of the two peers (on- and off- chip). Since a common communications infrastructure is missing, on-chip functionality may only be accessed from off-

* This work was supported by the Spanish Ministry of Education under Grants TIN2005-08719, and TEC2004-05205, and by Junta de Comunidades de Castilla-La Mancha under Grants PBI-05-049, and PBC-05-009.

chip components using an ad-hoc interface that exists only if it has been foreseen by the designer.

In this paper we will discuss the way our proposed methodology is able to tackle these and many other issues. We will first describe the remote method invocation model and how it is applied to NoCs, leading to a better separation of concerns between system design and system deployment. A thorough description of the mapping of this model onto an OCP based architecture will be done. We will them go through the different benefits and other consequences of applying such as flexible and homogeneous view of the system, including an analysis of the potential overhead.

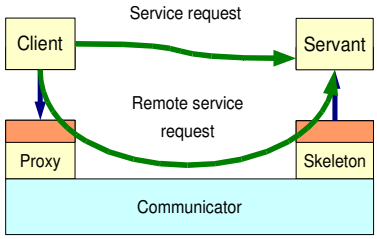

Figure 1. RMI provides an abstracted view of the communications channels.

2. Remote method invocation

Remote method invocation (RMI) is a synchronous communication model already popular in object-oriented distributed software systems [2][3][4]. RMI (see Figure 1) tries to emulate the behavior of a normal method invocation and dispatch of object-oriented programming languages. Objects are passive entities incarnated by some particular implementation (servant) waiting for requests to arrive somewhere in the network. A client requests a service from an object by issuing a RMI and blocks until the object replies. This is mostly the same semantics of the usual method invocation.

A common missconception is that such a synchronous model prevents parallel computations. This is not even the case of distributed software implementations, since both, client call and object dispatch may be handled by a different thread.

Figure 1 shows a simplified view of the logical structure. We may identify two roles for communicating objects. Servants (object implementations) are passive entities which share a set of services through a specific interface. On the other hand, clients may initiate a request following the interface exposed by the servant. A third component, the communications engine, act as a mediator between client and servant. Servant location, message routing, and network bridging are all responsibilities of the communications engine, or just communicator for short.

Actual connection of clients and objects with the communicator is made through entities that may be automatically generated from abstract object interface descriptions. In the client side there are *proxies*, which provide the illusion of always using local servants while they may actually reside on another computing device. On the object side there is a *skeleton*, which is resposible for adaptation of the specific servant interface and the messages in the network.

Objects and object implementations (servants) are kept as different concepts in order to easily provide many advanced features (object persistence, transparent replication, implicit activation, resource sharing, etc.). Therefore an object is an entirely abstract concept which is mapped to a specific servant by the communicator at method invocation time. Many popular software architectures fit in this model, such as CORBA[3], Java RMI[2], .NET Remoting[4], EJB[5], ICE[6], etc.

An independent communications engine component makes easier to transparently add a whole set of advanced features. One of the most relevant for this work is the ability to indirectly reference objects in order to achieve complete location transparency. That is, a method invocation may be issued to an object whose location is determined at run-time. Some important applications of this feature are the upgrading of on-chip components using off-chip modules, object migration to other locations, implicit object activation, automatic load balancing, etc.

The communicator keeps a mapping of objects and implementations (usually called *servants*)

through another component called *Object Adapter*. There is no need for a one-to-one mapping. It is possible to *incarnate* an object in more than one servants (e.g. fault tolerance through replication), and we may also reduce resource consumption by implementing a set of objects with a single servant. It is also possible to have objects which are not incarnated in any servant at a given time, and implicitly instantiate a servant at invocation time. We see this feature as a key factor for easier development of dynamically re-configurable systems.

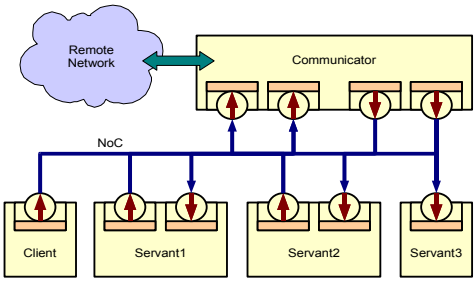

Figure 2: Simplified view of the middleware on a NoC.

The communicator effectively hides network details from clients and object implementations. Communicators on different platforms must agree on a shared protocol in order to allow remote interoperability, but components inside a single platform may optimize their internal communications, as we will see in the following sections. Each major distributed software middleware define a specific inter-communicator protocol (e.g. GIOP in CORBA and EJB, SOAP in .NET Remoting or WebServices, IceP in Internet Communications Engine, etc.).

These few simple abstractions provide a complete object-oriented framework for hardware modeling. Servants encapsulate functionality and state incarnating objects when they are needed. Proxies behave as low-cost references to remote objects, and skeletons translate network messages into local service requests. But designers of a NoC based system should also care about the overhead of this approach. A pure hardware design is much more static than a conventional software system and therefore it is hard to justify a significant overall overhead for rarely used dynamic object management features. In the following sections we will discuss in detail the implementation of the proposed middleware with minimum overhead as one of the major design constraints.

3. Global Communications System

The model described in section 2 does not depend on a particular communication protocol or data transport layer. Nonetheless we will describe a proposed mapping of this model to an OCP based architecture. The hardware implementation of the communications engine follows a logical structure similar to what is shown in Figure 2. Hardware objects are implemented by servants registered in an object adapter inside the communicator. The object adapter receives messages from the remote network and forwards them to the right servant. From a hardware point of view the communicator is a core which includes a number of general purpose services such as object adapters, remote servants, indirect servants, and communication primitives for inter-communicator interoperability. We will discuss the implementation of these features in the following subsections.

3.1 Hardware Objects

The first problem to be addressed is the physical implementation of objects. Traditional objects are dynamic entities, that may be created and destroyed in run-time. Their methods should be dispatched at run-time, rather than using static connections, in order to be able to implement inheritance and polimorphism. On the other hand, hardware is fundamentally static with the exception of dynamically reconfigurable systems.

An implementation of the object abstraction as it is known in the software domain would lead to unacceptable overheads due to the complex management mechanisms required. We limit the hardware support for objects to the bare minimum features which are obviously useful in the hardware domain and do not impose excesive overheads. We focus on dynamic creation and

destruction, separation of objects and implementations in order to maximize reuse, and low overhead object references. We explicitly avoid dynamic dispatching and run-time operation binding to avoid excesive overheads.

A hardware object provides just an encapsulation mechanism for some functionality (methods), shared among all of the instances of each class, and some per-instance private state data (attributes). Besides, methods are always considered static while attributes may change along the object lifetime. The semantics we use for objects is quite compatible to the traditional implementation of distributed objects (CORBA, EJB, .NET Remoting, etc.). The logical structure of a servant is shown in Figure 5 and will be described later.

The communicator provides an addressing mechanism to redirect object method invocations to servants. Each servant contains the physical implementation of all methods defined in a class and it is also responsible for managing per-object state data (attributes). This can be achieved with some storage elements (a local memory, a shared memory, a register file, etc.). Our proposed model do not impose a particular state storage method. Sometimes a shared memory will be used for state data storage. Other applications will require persistent storage in a FLASH memory. Sometimes we will instantiate a servant to implement a single object and therefore there is no need to keep several object states. It is even possible to define state-less objects. Since there is no way to directly access the state data from the clients we can be sure that the alternative chosen by the servant designer will not affect the rest of the architecture. Sharing servants among a set of objects is an easy way to reuse design components. If properly designed these shared servants do not prevent parallel execution of methods (e.g. by using a pipeline).

Each method invocation is characterized by a unique destination object identifier (*obj_id*), a unique operation identifier (method) and the set of parameters for that operation. Servants are quite similar to current IP blocks. Therefore legacy IP blocks may easily be integrated in our middleware based architecture incarnating a single object.

3.2 Proxies and skeletons

Interoperability is an additional requirement for servants not so commonly found in commercial hardware components. This is achieved by means of standardized interfaces called proxies and skeletons. Any method invocation must take place between a proxy and a skeleton. From the client point of view a proxy is seen as a private object implementation. It provides exactly the same physical interface. Servants on the other hand do not need to care about the location of clients. They just provide an object interface and export that interface through a skeleton.

For example, assume that the hardware middleware uses OCP as the communication protocol for some objects. Clients and servants are just cores that must behave as masters and slaves of the OCP bus. Therefore we need to translate the object interface (operations, parameters, ...) into OCP signals. On the client side proxies generate OCP messages from the invocations received. Skeletons receive OCP messages and translate them into signals of the object interface.

Clients and servants are not aware of the existence of an OCP bus. Indeed there is no need to use OCP. We propose OCP because it is a low overhead standard, but the communication protocol is abstracted by proxies and clients. This is an important observation for the designer since proxies and clients may be generated automatically from an abstract description of the object interfaces. Most modern distributed software middlewares already provide an interface description language for that purpose. Translating such interface descriptions into hardware requires the precise definition of a set of technology mapping rules giving a corresponding physical interface for each abstract object interface. For example, an operation may be mapped

into a one-bit input port. Designers and proxy/skeleton generators must adhere to that mapping rules.

Proxy and skeleton overhead may be negligible in most cases. A designer using OCP must already define an adapter to guarantee logical compatibility. Proxies and skeletons are equivalent to those adapters from the remote invocation point of view.

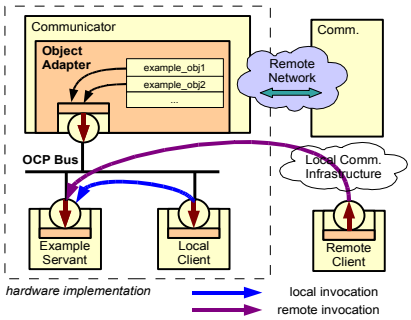

Figure 3: An object adapter makes on-chip objects available to off-chip components.

3.3 Object adapter

When the communications engine receives a remote invocation (from an off-chip component) it must first determine which servant should handle it. That is the role of a communicator component called the *object adapter*.

The incoming message contains an identifier of the destination object. This identifier is used to index a table (*Active Object Map*, see Figure 3) which relates each object identifier with the corresponding servant. Any object whose identifier is not registered in the active object map cannot be accessed from the remote network. On the other side, servants not incarnating any object are considered inactive, but they may still be used locally.

Communication between object adapter and servants is performed as described in section 3.2. The object adapter holds a collection of proxies used to contact the destination servants. As stated before, several objects may be incarnated by a single servant (entries in the *Active Object Map* pointing to the same servant), or a single object may be replicated on several servants (replicated entries in the *Active Object Map*). Note that the middleware does not guarantee replication transparency on the servant side. Servants are responsible for consistency management.

Efficiency and low overhead are two major concerns in a hardware communications framework. The model described allows run-time registration of new servants in the *Object Adapter* and dynamic control of remote accessibility of new objects by adding new entries in the *Active Object Map*. Both are feasible operations in hardware. Registering new servants may require adding new proxies to the object adapter if they are not known in advance. That feature may be provided by dynamically reconfigurable devices.

Most of the communications infrastructure is either generated automatically (skeletons and proxies) or provided by the hardware communicator. The system designer may then postpone decisions about the system deployment until the latest stages of the design flow.

3.4 Local Objects and Remote Objects

As described above proxies and skeletons provide network transparency. In most distributed software middlewares proxies are opaque entities which encapsulate local or remote access to the destination object. Local accesses may be easily optimized by direct translation of remote invocations to standard method calls. These are usually called *co-located servants*. Our proposed architecture also allows direct local object interaction, without the help of the communications engine.

Hardware proxies are not required to store any information on the physical location of remote objects. The communicator provides special purpose remote servants which translate invocations to remote objects into messages at the remote network (see Figure 4). These *remote servants* may also be automatically

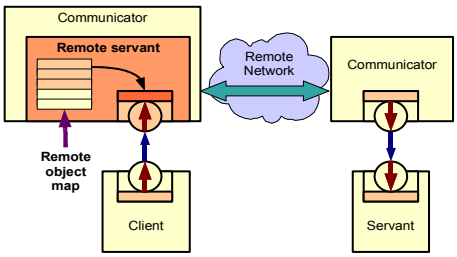

Figure 4: From the point of view of the proxy there is no difference between local and remote objects.

generated.

We consider local invocations by default handling the remote objects as a special case. Software middlewares do exactly the opposite. Each remote servant holds a *Remote Object Map* which relates local bus addresses with global object identifiers as used in the remote network. In contrast to software middlewares our approach introduces almost zero overhead for local communications, which is a major concern for hardware designs.

A middleware with the previously described mechanisms already allows a large degree of location transparency. The design of the different entities in the system is independent of the physical location of each one. It is even possible to implement some entities in software without modifications to the other entities. But modern distributed software middlewares allow run-time location of objects. This is specially interesting for hardware designs because it would allow component upgrading even after fabrication.

We provide this level of location transparency through another communicator component, an specialized *indirect servant*. Proxy to servant protocol must be augmented to include a *redirect* message. At invocation time the indirect servant examines where is actually located the destination object and emits a *redirect* message containing the actual address of the destination object. When a proxy receives a *redirect* message it must retry the invocation using the received data. This kind of proxies are a bit more complex but they are only needed when a higher level of location transparency is required. Given that both proxies and skeletons are automatically generated, the designer may evaluate the trade-off between flexibility and overhead to suit his needs.

4. Experimental results

Let us illustrate the discussion in the previous sections with a little example. Figure 5 shows the UML definition of the example_class from which some objects will be derived. The class only contains the value attribute and two methods to read and write the attribute. This extreme simplicity has been choosen for the sake of clarity, since the concepts presented here could be generalized to more real and hence much more complex situations.

We will assume that the system architect already decided to make hardware implementation of the objects of this class remotely accessible from anywhere in the network. So the implementation will consider two different situations for the invocation of the methods: local and remote communication.

As already mentioned in the description of the comunication engine, that will not always be the case for all hardware objects, since many of them will only be useful for local computations. In that case the servant implementing the objects will only be accesible through the local communication infrastructre. Besides, we will also assume the use of OCP to provide the transport protocol between local comunicating entities. Remote comunication can be provided through any kind of network interface. Figure 5 shows the structure of the example servant. We can distinguish two main parts, one related to the implementation of the objects of the example class, and the skeleton that will provide access to the methods of the class. Objects are accessed through a component whose interface mimics that of the object plus some extra information such as the object identification (*obj_id*). The state of all the objects is stored separately and can be addressed through the object indentification. The second part of the servant is the skeleton that translates the local comunication protocol to an operation invocation. One important thing to note is that skeletons are automatically generated from the interface definition of the object (once the mapping rules for the local protocol have been defined). Since local comunications are performed over an OCP logic bus, the skeleton is in fact an OCP to the object interface slave bridge.

For a client to perform any invocation, a proxy with the target object interface must be

implemented. The proxy mirrors the skeleton, providing the inverse, object interface to OCP, translation (see Figure 6). The proxy is also generated automatically from the same object interface description than the skeleton, so it can be easily optimised for the application. It could be possible, for example, to include in the proxy only those operations that will be invoked by the client.

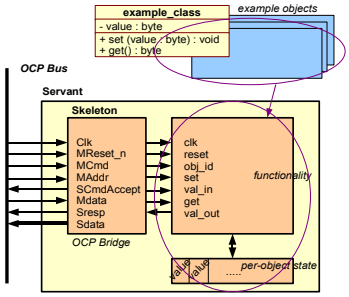

Figure 5: Simple servant handling multiple instances.

As described in section 3 remote invocations will be handled by a specialized communicator component named the *object adapter*. The object adapter is composed of two parts. 1) On one side it holds a finite state machine for message handling (parsing and building), and for marshalling and unmarshalling of the available data types in the network. The precise definition of this state machine depends on the selected middleware protocol (GIOP for CORBA interoperability, IceP for ICE interoperability, etc.) but there is no need to re-design it for each application. 2) After successful reception of a message, the object adapter locates the destination object by means of a lookup table and routes the raw data through the corresponding proxy.

The process for a local invocation is the following. 1) The client activates the corresponding signals of the local proxy component. 2) The proxy translates the invocation into a an OCP master comand that establishes a point to point comunication between the client and the servant (whose address is coded in the proxy). 3) The OCP slave (the skeleton) receives the command through the NoC that is translated into the proper signal activations of the objects component interface. 4) The operation is executed using the corresponding state information.

When an on-chip component invokes a method of an off-chip object then a remote servant comes into play. The remote servant redirects the incoming data through the network using the appropriate marshalling and protocols. It is obvious that the remote servants share a lot of capabilities with object adapters and therefore middleware implementors may share communicator resources between these two components.

As shown in this example, proxies and skeletons are mapped to simple OCP adapters. Therefore the system will even work in the absence of a communicator. In any case, the system designer may evaluate at any point of the design flow the tradeoff between minimum cost implementations and flexibility of the deployed system.

5. Implications of a Hardware Middleware

As described above, a hardware middleware introduce a relatively small set of concepts and the physical implementation is straightforward. One may be tempted to think that the architecture we propose is just a minor evolution of traditional hardware design and miss the major strenghts of our approach.

From the point of view of the design methodology, a fully functional simulator of the whole system may be developed in the early stages of the design on a single computer using a standard middleware. Afterwards, in every step of the design cycle there is a complete functional system. Simulated software components will directly interact with a partial NoC design without any special consideration, leading to easier system debugging. System deployment and global system partitioning will constitute a new orthogonal role.

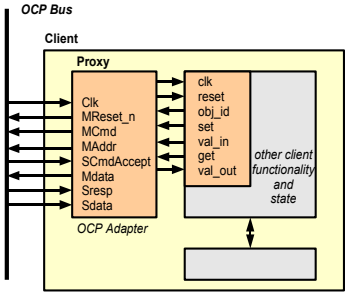

Figure 6: Proxy using an OCP master to interface the NoC.

There has been an intense debate on which place should the operating system occupy in system design. Proposals range

from a single HW/SW operating system for the whole system to the concept of distributed nanokernels. Fortunately the communications engine is independent of the operating system. If required, it may even be implemented on top of the hardware-software middleware leveraging the transparency features of the communicator.

From the point of view of the customer, a hardware middleware allows dynamic on-line upgrades of systems without disturbing their normal operation. A new implementation of a given module may be added to the system by just adding an entry in an indirect servant.

A hardware middleware opens a whole new range of possibilities we did not consider in this paper due to space constraints. For example, it is feasible to automatically generate pieces of hardware to automate state storage and recovery. This would be the first step in the development of transparent hardware object migration, specially interesting for dynamically reconfigurable distributed systems.

The introduced overhead depends on the final deployment of the whole system. For example, many hardware objects should not be accessed from the remote network. In this case there is a minimum overhead since we use the same logical buses or NoCs as middleware-less designs.

Efficiency considerations may lead to even deploy point-to-point communications between some pairs of clients and servants. Besides, NoCs used for proxy to servant communications may be optimized as usual. For example, it is possible use a segmented bus, or a complete switched NoC with OCP interfaces. This is entirely orthogonal to our approach.

6. Conclusion

The middleware based methodology proposed in this paper introduces very few abstractions and may easily be integrated in current design flows. In spite of its simplicity it provides a vast range of benefits which are briefly summarized below.

1) It provides a low-cost hardware object-oriented framework which is quite handful to integrate system-level object oriented modeling such as UML.

2) It provides a high degree of transparency for system designers. NoC transparency is achieved with proxies and skeletons, remote network transparency is provided by the *remote servant* and the *object adapter*. Full location transparency is provided by the *indirect servant*.

3) It also provides some advanced features such as basic support for load balancing, fault tolerance through replication, and transparent instantiation of dynamically reconfigurable hardware. These mechanisms set up the basis for object persistence and object migration.

We believe all these features cooperate nicely to provide better orthogonalization of concerns and an integrated and homogeneous view of the whole NoC based system. The introduced overhead may easily be tuned at late stages of the design flow.

7. References

[1] *Open Core Protocol Specification, Release 2.0*, OCP-IP Association 2003.
[2] *Java™ Remote Method Invocation Specification*, Sun Microsystems, Inc. 2003.
[3] *The Common Object Request Broker: Architecture and Specification, Revision 2.6.1*, Object Management Group 2002.
[4] Piet Obermeyer, and Jonathan Hawkins. (2001, July). *Microsoft .NET Remoting: A Technical Overview* [Online]. Available: http://msdn.microsoft.com/ library/en-us/dndotnet/html/hawkremoting.asp
[5] *Enterprise JavaBeans Specification, Version 2.1*, Sun Microsystems 2003.
[6] Michi Henning, and Mark Spruiell, *Distributed Programming with Ice*, Palm Beach Gardens, FL: ZeroC, Inc. 2005.